An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries and Case Study on PRESENT – Extended Version

Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, Axel Legay

An Automated Formal Process for Detecting Fault Injection Vulnerabilities in Binaries

and Case Study on PRESENT – Extended Version

Thomas Given-Wilson, Nisrine Jafri, Jean-Louis Lanet, and Axel Legay
Inria
Rennes, France
Email: {thomas.given-wilson, nisrine.jafri, jean-louis.lanet, axel.legay}@inria.fr

***Abstract*—Recently fault injection has increasingly been used both to attack software applications, and to test system robustness. Detecting fault injection vulnerabilities has been approached with a variety of different but limited methods. This paper proposes a general process without these limitations that uses model checking to detect fault injection vulnerabilities in binaries. The efficacy of this process is demonstrated by detecting vulnerabilities in the PRESENT binary.**

I. Introduction

Recently fault injection has been increasingly used both as a method to attack software applications, and to test the robustness of software systems. Many systems are particularly vulnerable to fault injection attacks due to operating in hostile environments, i.e. environments where an attacker may be able to perform physical attacks on the system hardware. Many such attacks have been demonstrated on a variety of systems, showing that different kinds of faults can be injected into various devices [1], [2], [3]. Attacks can also be achieved through software alone and do not require attacking the hardware directly. One recent example of this is the row hammer attack [4] that has been exploited to perform various attacks [5], [6].

The wide variety of fault injection attacks and possible impacts upon a system make it impossible to prevent software from failing under all possible attacks [3]. Thus, recent work has approached the problem of fault injection by limiting the scope of attacks, or limiting the kinds of vulnerabilities analysed [7], [8], [9], [10], often requiring specialed equipment.

This paper proposes an automated formal process for the detection of fault injection vulnerabilities in binaries. In particular, a process that can account for many different kinds of fault injections and that does not require extensive hardware or specialised equipment. This process is achieved by simulating fault injection attacks upon the executable binary for the given software, and then using model checking to determine whether or not the simulated fault injection attack violates properties the software should maintain.

This paper presents an automated process for detecting vulnerabilities in binaries using model checking. The process begins with the *executable binary* that represents the program to be considered. The validation of the binary involves checking various *properties* using model checking to ensure the binary meets its specification. Fault injection attacks are then simulated on the executable binary, producing *mutant binaries*. The properties are then model checked on the mutant binaries. A difference in the result between validating and checking the properties indicates a vulnerability to the fault injection attack that was simulated.

This process provides a general approach that can support detecting a wide variety of fault injection vulnerabilities in binaries by varying the fault model of the fault injection. The strengths of this approach include the following. By operating directly upon the binary, fault injection vulnerabilities that cannot be detected in source languages or intermediate representations can be detected [11]. Formal methods, here model checking, ensure the rigour of the analysis and so ensure that fault injection vulnerabilities that are detected are real and not false positives. An automated process can easy to iterate over various fault injection models and approaches, and also allows broad, or even complete, coverage of possible fault injection attacks. Combining automation, broad coverage, and formal methods, allows the process to make strong guarantees about the vulnerability of a system that has been analysed.

To demonstrate the efficacy of this process, this paper includes a case study of applying the process to the PRESENT encryption algorithm [12], [13] with various

fault models. PRESENT is a lightweight encryption algorithm designed to be used on embedded devices, thus making it an ideal choice to consider security critical software in hostile environments. The process proposed here was applied to the PRESENT binary with five different fault models for a total of approximately 5700 experiments. These fault injections yielded a number of infinite loops, and crashes, but also 9 vulnerabilities where the encryption algorithm is completely bypassed.

The key contributions of this paper are as follows.

- Describing a general process that allows automated detection of fault injection vulnerabilities in binaries.
- An implementation of the process that allows easy automation with existing tools.
- A case study demonstrating the efficacy of using the process on the PRESENT algorithm.
- The identification of 9 fault injection vulnerabilities in the PRESENT algorithm that bypass encryption.

The structure of the paper is as follows. Section II presents a simple motivating example that is used to clearly demonstrate the techniques. Section III recalls background on fault injection attacks, model checking, properties, and PRESENT. Section IV details the process proposed in this paper. Section V discusses the implementation and tools used to achieve the process. Section VI presents illustrative results for the motivational example. Section VII applies the process to the PRESENT algorithm with five fault models and analyses the results. Section VIII discusses related work. Section IX concludes and discusses future work.

II. Motivating Example

This section presents a motivating example that is used to: illustrate the concepts and process, and for the experimental results of this paper. The example is of code that checks a PIN supplied by a user when authenticating to use a credit card.

Consider the code in Figure 1 that checks the value of a candidate PIN entered by a user when authenticating to use a credit card. Prior to this code fragment the true PIN `PINTrue` is assumed to be defined and intialised with the true PIN value. Similarly the candidate PIN `PINCandidate` is defined and initialised with a value input by the user. Further, both PINs are checked to be the same length and this length is defined to be their size `PINSize`. The first three lines initialise variables to be used in the code fragment shown here. The `grantAccess` variable is used to indicate whether

```
bool grantAccess = false;                          1
bool badValue = false;                             2
int i = 0;                                         3
while (i < PINSize) {                              4
  if (PINCandidate[i] != PINTrue[i]) {             5
    badValue = true;                               6
  }                                                7
  i++;                                             8
}                                                  9
if (badValue == false) {                           10
  grantAccess = true;                              11
}                                                  12
```

Fig. 1: Motivating Example Code

or not to grant access after checking the candidate PIN against the true PIN, intialised to `false`. The `badValue` variable is used to detect when digits of the two PINs do not match, also intialised to `false`. The variable `i` is used as a an iterator to progress through the digits of the PINs, intialised to `0`. The next six lines of the code are a loop that iterates through the digits of the candidate PIN and true PIN, incrementing `i` on line eight, and bounded by the PIN size `PINSize`. Line five checks the `i`th digit of the candidate PIN against the `i`th digit of the true PIN. On line six, if the digits differ, then `badValue` is set to `true` to indicate that (at least one) digit of the two PINs are not equal. At the end, on line ten the conditional checks that no bad values have been found, and grants access if this is the case.

Now consider when `PINSize = 4`. By changing a single bit, an attacker could change the value of `PINSize` from `4` to `0`. (This succeeds since `4 = 0...0100` in binary, and changing the `1` to `0` yields `0...0000`.) Observe that this would bypass the loop since `i < PINSize` (i.e. `0 < 0`) would not hold, and therefore the checking of any digits of the candidate PIN. Thus, the example is vulnerable to this kind of 1-bit fault injection attack (as well as several others that will be introduced later).

The above paragraph describes a fault that can be injected into the executable binary that would allow the attacker to gain access even without the correct PIN. This paper proposes and explains a process that can be used to detect such fault injection vulnerabilities (Section IV). An implementation of the process is detailed in Section V, and this implementation is used to obtain the experimental results (Section VI). The motivating example is used as the basis for the experimental re-

sults, including detecting the fault injection vulnerability described above.

III. Background

This section recalls key concepts and information useful to understanding the rest of the paper. These are divided into four main areas: fault injection, model checking, properties, and the PRESENT algorithm.

A. Fault Injection

Fault injection can be considered as an attack, when an attacker targets the hardware of a system to create an exploitable error at the software level. The goal of such an attack is to cause a specific effect at the hardware level, that in turn creates an exploitable change in the software behaviour. The rest of this section recalls key points regarding the kinds of fault injection attacks considered here.

The hardware effect of a fault injection attack is described through a *fault model* that specifies the nature and scope of the induced modification. Typically such attacks are achieved by changing a value stored in the hardware, such as changing the value of a whole byte [3]. Such hardware effects generally focus on the kind of fault that can be created rather than the effect this has on the software.

Simulating fault injection attacks in an experimental environment can be done in two ways: reproducing the attack on the hardware, or simulating the attack with software. Reproducing the attack using hardware technology is relatively difficult and expensive, since specialised hardware must be used to inject the fault (e.g. using a laser [7], or electromagnetic pulse [8]). Comparatively, simulating with software is easy and cheap because this requires only modification to the executable binary and no specialised hardware. Since the goal in this paper is to develop an efficient process that can be implemented with a software tool chain, the rest of this paper will only consider software simulations.

Software simulation attacks can also be classified into two kinds of fault injection attacks, *run time* and *compile time*. Run time fault injection attacks are those that occur only while the code being attacked is being executed. Compile time fault injection attacks are those that occur at any time starting form compilation of the code, and up until just prior to execution. This paper considers only compile time fault injection attacks since this captures many run time faults as well and also builds towards future work (see Section IX-A).

B. Model checking

Model checking is a formal method for determining whether properties hold on a model [14]. Model checking has the advantage that all possible states of the model are considered, and so is guaranteed to be able to answer whether or not a property holds for a given model.

The *model* is a representation of the program or system being considered. A good model is able to represent all the possible states and transitions that the program can achieve. In this work, the model represents an executable binary program.

The cost of model checking comes in the potential exponential complexity used to consider all the possibly infinite states of a model. However, for limited models, model checking is highly efficient and precise. Further, various approaches have been used to make model checking efficient even for large and complex models (or programs) [15].

Bounded model checking is a refinement of model checking that alleviates some of the issues with possibly infinite complexity by bounding the checking [16]. The key idea in bounded model checking is to put a bound on parts of the model that could be infinite (or at least extremely large). For example, checking a program with a loop, going through hundreds or millions of iterations could be very costly for model checking. However, bounded model checking of such an example could limit the number of times to iterate through a loop. Thus, bounded model checking allows limits to be placed upon such potentially unbounded aspects of model checking.

C. Properties

To perform model checking requires specifying the *properties* to be checked upon the model. There are two main kinds of properties that can be checked *safety*, and *liveness* [14]. Safety properties are used to express that certain propositions hold when they are encountered. Liveness properties express that propositions hold over some temporal dimension. This paper only considers safety properties since these are clearer, more intuitive to represent, and sufficient to illustrate the feasibility of the process. Liveness properties can also be checked in a similar manner, although this is not presented in this paper, for further details see the discussion in Section IX-A.

Safety properties can be expressed by simple propositions that can be annotated into the code of the program being considered. Generally such properties support: negation, equality, inequality, conjunction, and disjunction. These can be defined upon values or states

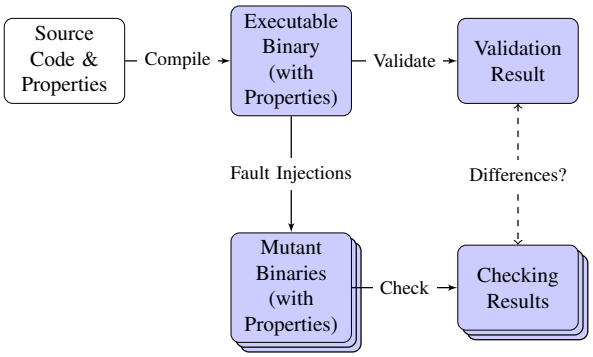

Fig. 2: Process Diagram

and values within the model (and thus the binary being checked).

D. The PRESENT Algorithm

PRESENT [12], [13] is a lightweight block cipher designed for use on low power and CPU constrained devices. The PRESENT algorithm consists of 31 rounds of a Substitution-Permutation Network (SPN) with block size of 64 bits. The canonical implementation of PRESENT[1] supports key lengths of 80 or 128 bits. The core encryption algorithm is the same for both 80 and 128 bit keys.

The version of PRESENT analysed here is the canonical version in C for 32 bit architectures (size optimised, 80 bit key) with minor modifications to change loop types (due to limitations in the tools used, see Section IX-A for further discussion of these).

IV. Process

This section details the process presented in this paper for detecting fault injection vulnerability in binaries.

An overview of the process is as follows (and shown in Figure 2). Prior to starting the process, the source code, and the properties represented by annotations within the source code, must be defined. The preparation step for the process is then to compile the source code and properties to produce an executable binary in a manner that preserves the properties as annotations. The properties are validated to hold on the executable binary using model checking. The executable binary is then injected with simulated faults to produce mutant binaries. The properties are then checked upon the mutant binaries again using model checking. A difference in the results of validation and checking the properties indicates a vulnerability to the simulated fault injection. The rest of this section details this process.

The choice to start the preparation with the source code and not the binary is made for illustrative clarity and ease of use for the software developer, since defining properties over binaries is more arduous. However, most aspects of the process do not rely upon this choice, and future work is to be able to start directly from the binary (further discussion in Section IX-A)

Since this paper considers fault injection attacks upon the binary it is necessary to compile the source code (and here properties) into an executable binary. This executable binary represents the software application that would be executed by the system in practice. Thus to simulate fault injection attacks on the actual system, the executable binary must be used in the simulation. For the process here the compilation must maintain the properties, and so compilation must maintain the properties as annotations in the executable binary.

The properties are validated to hold upon the executable binary using model checking. This is done to ensure that the executable binary does indeed meet the specification of the properties. If there is some other error in the source code or compilation, this can be detected here and not be (incorrectly) attributed to fault injection vulnerability.

Next is the simulation of the fault injection on the executable binary to produce mutant binaries. This simulates the actual fault injection attacks and produces mutant executable binaries that represents the executable binary after the attacks have been effected.

Lastly, the validation results from model checking the executable binary are compared with the checking results from model checking each mutant binary. Differences here indicate that the fault that was injected yields a change in behaviour that violates the properties, and so could be exploited by an attacker.

V. Implementation

This section presents the implementation of the process from the previous section. The implementation here exploits currently available tools where possible, despite some having significant limitations. This choice was made in order to focus upon a simple and feasible implementation of the process. For discussion of the limitations of the tools used in the implementation here, please refer to Section IX-A.

The presentation here is highly detailed to allow easy reproduction. This choice was made to maximise clarity and opportunity for others to implement the process for themselves and conduct their own experiments. The rest

[1] Available at http://www.lightweightcrypto.org/implementations.php

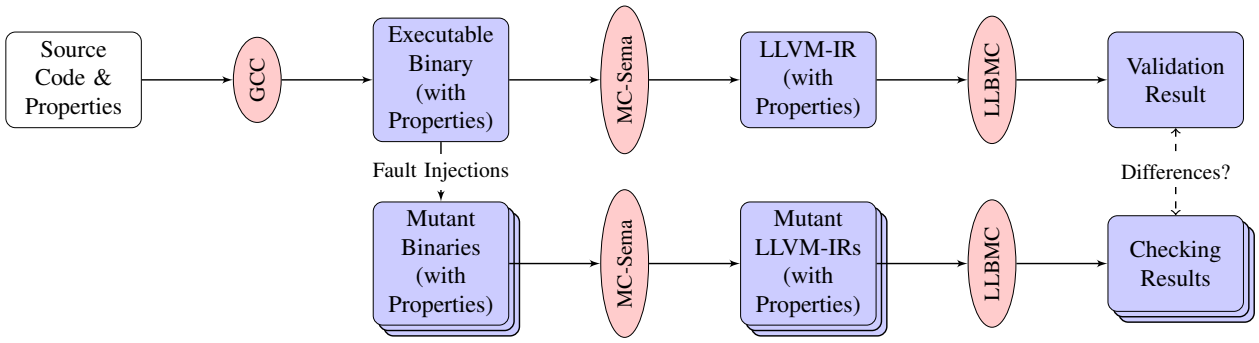

Fig. 3: Implementation Diagram

of the section explicitly details how to implement the process with the chosen tools.

An overview of the implementation is as follows, and shown in Figure 3. The implementation begins with the source code written in the C language and the properties represented in the source code by assert statements. The source code and properties are compiled to an executable binary by the GNU C Compiler (GCC) [17]. The executable binary (including the properties contained within) is transformed into an *intermediate representation* in Low Level Virtual Machine Intermediate Language (LLVM-IR) by the Machine Code Semantics (MC-Sema) tool [18]. The properties are then checked on the intermediate representation using the Low Level Bounded Model Checker (LLBMC) [19], [20]. The executable binary is manually edited to produce the mutant binary file according to the fault model chosen. The steps to model check the properties on the executable binary are then repeated for the mutant binary. Finally, the results of model checking the executable binary and the mutant binary are compared for differences. The rest of this section details this implementation.

A. Source Code & Properties

The implementation starts with the source code written in the C language, including the properties to be validated and checked that are expressed as assert statements in this source code. For example, the source code of the motivating example (see Section II) could have the following property (recalled from Section III-C)

```
__llbmc_assert(i == 4);
```

that the loop counter `i` reaches `4` inserted between lines 9 and 10. This would check that `i` reaches the value `4` before doing the conditional to test whether access should be granted on lines $10-12$.

B. Compilation

The compilation from source code to executable binary for this paper is done with GCC. Note that here a listings file is also generated with annotations that will be exploited to do the fault injection later. The following is the command used to compile with GCC for this paper.

```
$gcc -m32 -ggdb -c -Wa,-a,-ad -o test.o
     test.c > test.lst
```

Here `-m32` specifies compiling for 32-bit architecture. The `-ggdb` argument includes debugging information that will be used to help translate the intermediate language later in the implementation. The `-c` argument indicates to compile and assemble the source code, but do not link (this simplifies the scope of checking since no library code is linked at this stage). The `-Wa,-a,-ad` argument specifies annotations to output that will be used later to do the fault injection (`-a` to turn the listing on, `-ad` to omit unnecessary debug information). The `-o` is used to specify the output file (`text.o`) for the executable binary. Here `test.c` is the source file with properties. Lastly, `> test.lst` outputs the annotations used later to do the fault injection into the file `test.lst`.

Note that the above command preserves the assert statements along with the debugging information, so these can be exploited in later stages.

C. Intermediate Representation

The translation from executable binary to LLVM-IR is done by MC-Sema in two stages. The first stage uses the executable binary to generate a Control Flow Graph (CFG). The second stage uses the CFG to generate the LLVM-IR.

Executable Binary to CFG: The first stage is done by the `bin_descend` tool (included within MC-Sema) using the below command.

```
$bin_descend -march=x86 -d
```

```
    -func-map="test_map.txt"
    -entry-symbol=checkPIN -i=test.o
```

Here the `-march=x86` argument specifies X86 architecture. The `-d` flag enables output of debugging information, used in later stages of the implementation. The `-func-map= "test_map.txt"` argument informs of the file (`test_map.txt`) that contains specifications of externally referenced functions (e.g. `__llbmc_assert 1 C N` to indicate that the function `__llbmc_assert` has 1 argument, `C` to represent the calling convention here is for CleanUp to clean up the stack after the function call, the `N` to mention that the function has a return). The `-entry-symbol=checkPIN` argument indicates the function name of the entry point into the code, here the `checkPIN` function. Lastly, `-i=test.o` indicates to input from the file `test.o`.

CFG to LLVM-IR: The second stage is to translate the CFG to LLVM-IR. This is done by the `cfg_to_llvm` tool also included in MC-Sema. The command to achieve this is shown below.

```
$cfg_to_bc -mtriple=i686-pc-linux-gnu
    -driver=test_entry,checkPIN,0,return,C
    -o test.bc -i test.cfg
```

Again `-mtriple=i686-pc-linux-gnu` indicates the X86 architecture (on linux). The argument `-driver=test_entry, checkPIN,0,return,C` defines the entry point in the generated LLVM-IR to be `test_entry` and this should correspond to the entry point symbol `checkPIN` in the CFG, the `0` represent the argument count, the `return` to specify that the function has a return, and finally the `C` represents the calling convention. As usual `-o` indicates the output file name (here `test.bc`). Lastly, `-i` is the input file name (here `test.cfg`).

D. Model Checking

The model checking of properties on the intermediate representation is done by LLBMC. The command used here to model check with LLBMC is as below.

```
$llbmc -function-name=test_entry
    --ignore-missing-function-bodies
     --max-loop-iterations=20
     -only-custom-assertions test.bc
```

The argument `-function-name=test_entry` makes certain that LLBMC checks the specified function (and not others). The `--ignore-missing-function-bodies` argument is used to ignore missing functions, such as those that were not linked and so do not appear in the LLVM-IR. The argument `--max-loop-iterations=20` specifies bounds for the model checking, here limited to 20 loop iterations. The `-only-custom-assertions` argument forces LLBMC to only check the properties specified, and not other default properties. Lastly, `test.bc` is the input file.

```
Result:
=======
No error detected.
```

Fig. 4: Property 1 : executable binary verification result

Note that the above steps from executable binary to model checking can be repeated (with changed file names) for the mutant binary, and so will not be repeated below.

E. Fault Injection

The fault injection attack that takes an executable binary and yields a mutant binary is done by manually editing the file. To achieve this the listing file generated by GCC during compilation is exploited to find which locations in the executable binary correspond to the source code, and thus easily understand the semantics so as to be able to simulate specific fault injection attacks. Once the locations can be found with the listing, the binary can be modified using any binary editing tool and the modified one saved as the mutant binary.

F. Detecting Vulnerability

Once the results of model checking have been produced for both the executable binary and the mutant binary, fault injection vulnerabilities can be detected when these results differ. In Figure 4 the output of LLBMC is shown for when all the properties hold. By contrast Figure 5 shows the LLBMC output when a property is violated. Note that due to compilation to binary and then translation to LLVM-IR, LLBMC is unable to gather sufficient information to produce a useful trace of the property violation.

VI. Experimental Results

This section presents experimental results obtained by applying the process using the implementation. For clarity, all the attacks and properties presented here use the motivating example (of Section II). The experimental results show that by using the process presented here (and the implementation detailed in Section V) a variety

```
Result:
=======
Error detected.

Error synopsis:
===============
Assertion failed: Custom assertion (
    assert or __llbmc_assert) does not
     hold.

Error location:
===============
Error occurs in basic block "
    block_0x10b" of function "sub_0".
No debug information available.

Stack trace:
============
#0 void @sub_0(%struct.regs* %0)
#1 i32 @demo_entry()
```

Fig. 5: Property 1 : mutant binary verification result

of fault injection vulnerabilities can be detected. The rest of this section presents illustrative 1-bit fault injection attack examples building to the general solution that can detect many kinds of fault injection attacks, and concludes with a variety of different kinds of fault injection attacks to illustrate generality of the process and implementation.

Attack 1: The first attack to consider is the attack detailed in Section II where changing a single bit changes the `PINSize` variable from `4` to `0`. Recall that since the loop that checks the digits of the PINs iterates from `i = 0` to `i < PINSize`, that the loop will be skipped (since `0 < 0` does not hold). This fault injection attack can be exploited by the attacker since the PINs are never checked against each other. Therefore, any candidate PIN will lead to access being granted, and so the attacker can use this fault injection attack to gain access (even when the candidate PIN they supply does not match the true PIN).

Property A: A simple property to detect such an attack would be to ensure that `i` reaches `4`. This can be achieved by taking the following property (recalled from Section III-C):

```
__llbmc_assert(i == 4);
```

inserted between lines 9 and 10.

With this property added to the source code the process was repeated with attack 1. The results of model checking the mutant binary reveal that the assertion was violated and thus the model checking result different form the validation result. Thus, vulnerability to the first fault injection attack was detected.

Attack 2: An alternative 1-bit fault injection attack that has the same effect as Attack 1 is to initialise the value of the variable `i` to `4`. This will again grant access even when the two PINs differ since the loop will be bypassed as before (this time since `4 < 4` does not hold). Observe that since `i` is intiailised to `4` this fault injection attack should not violate the assert statement of Property A. This can be exploited by the attacker in the same manner as Attack 1 to gain access with a candidate PIN that does not match the true PIN.

The process was repeated with Attack 2 and Property A. As expected, the fault injection vulnerability was not detected.

Property B: The above result illustrates that the choice of properties need to consider the behaviour of the program rather than focus on particular variables that are incidental to the program's execution. Thus, a property that captures the idea that unequal PINs should never lead to access being granted could be defined as follows (also recalled from Section III-C).

```
__llbmc_assert(    !(PINCandidate != PINTrue)
                || grantAccess == false);
```

where this property is inserted into the motivating example code after line 12. This property expresses that if the two PINs are different then the access is not granted.

The process was then repeated for both Attack 1 & 2, and with Property B. As expected Property B was able to detect both fault injection vulnerabilities represented by Attacks 1 & 2. This shows that considering the behaviour is more important than considering the variables used to achieve the behaviour. That is, properties should consider `PINCandidate`, `PINTrue`, and `grantAcces` rather than `i` or `badValue`.

Attack 3: Observe that Property B detects attacks that allow access when the PINs are not equal, but does not consider when the PINs are equal. An alternative attack could be to deny access even to a user who knows the correct PIN. Consider the 1-bit fault injection attack that changes the value of `true` (represented in binary by `0...01`) to `false` (represented in binary by `0...00`).

Now consider this attack upon the motivating example line 11, changing `grantAcces = true` to be

```
grantAccess = false;
```

and thus preventing any access even when the PINs are equal.

The process was then repeated using this new Attack 3 with Properties A & B. As expected neither property was able to detected this attack. Property A failed since the attack does not effect the iterator `i`. Property B failed since when the PINs are equal no further behaviour is considered.

Property C: To also account for Attack 3, the original Property B needs to be extended to also consider the behaviour when the PINs are equal. Indeed, the ideal behaviour of the code can be represented by the following property.

```
__llbmc_assert(
     (   !(PINCandidate != PINTrue)
      || grantAccess == false)
  && (   !(PINCandidate == PINTrue)
      || grantAccess == true));
```

This ensures that when the PINs are unequal access is not granted, and when the PINs are equal then access is granted. Property C is added to the motivating example in the same place as Property B would be; after line 12.

The process was then repeated with all three Attacks (1, 2 & 3) using Property C. As expected Property C was able to detect all three fault injection attacks.

A. Other Attacks

This section considers several more fault injection attacks in less detail than those above. These include several more 1-bit fault injection attacks, and then other kinds of attacks, culminating in an attack that can only be effected in the binary and not in the source or by "compiling" directly to an intermediate representation.

There are several other 1-bit fault injection attacks that can be performed against the motivating example. Such attacks include changing the initialisation value of variables such as `grantAccess` and `badValue`, e.g. at line 6 changing the initialisation `badValue = true` to instead be `badValue = false`. These are all detected by at least Property C, if not also Property B.

A different kind of attack that targets the control flow of the program is to change the target of a jump instruction. For example, the jump from the conditional on line 5 of the motivating example, could change from skipping the following instruction on line 6, to always executing line 6. Thus `badValue = true` (line 6) would always be executed, regardless of the outcome of the conditional. This can be done by modifying three bits (or one byte) of the target (relative) address of the jump, from `0000 0111` to `0000 0000`. This attack was successfully detected by Property C (but not Properties A or B).

A more significant attack on the behaviour of an instruction is to simply change the instruction to a NOP (non-operation). Consider in the motivating example when a fault injection changes the instruction that represents line 6 `badValue = true;` to a NOP. This requires modifying 4 bytes (on the X86 32-bit architecture used here). The change would allow access for any candidate PIN (by never recording differences to `badValue`). This attack was successfully detected by Properties B and C (but not Property A).

Instead of changing a subtle behaviour like a jump, or simply wiping an instruction to a NOP, another fault injection attack is to change the instruction type, e.g. changing a `CMP` instruction to a `MOV` instruction. This can be done on the `CMP` instruction that compares the values `PINCandidate[i] != PINTrue[i]` on line 5 of the motivating example. This requires modifying 3-bits of the executable binary, from `0011 1011` to `1000 1011`. The result of this change is that the following line that sets `badValue` to `true` will always be executed. This change prevents access even when the correct candidate PIN is provided, similar to Attack 3. As expected, this attack is successfully detected by Property C (but not Properties A or B).

One attack that is of particular interest here, is an attack that can be represented in the executable binary but not in source code or from "compiling" the source code to an intermediate language, such as C and LLVM-IR, respectively. An example of this kind of attack is the modification of the return value stored in the return register (`eax` on X86 architectures). This kind of attack can be simulated and detected using the process and implementation here.

To properly handle this attack requires a function call, and so the motivating example is modified by placing the code in Section II inside a function that returns `grantAccess`. The attack works by altering the returned value from this new function. The return value is stored in the `eax` register, for the motivating example this is handled by the binary operation corresponding to the assembly instruction below.

```
mov    eax, DWORD PTR [ebp-0x8]
```

The attack is then to change the value loaded into `eax` so that the function behaviour is changed. For example, by changing the value `0000 1000` to `0000 1100` the returned value of `grantAccess` can change from `False` to `True`, so the access will be granted even if the two PINs are not equal. (Note that if the returned value is already `0000 1100` then this can be ignored, or the value changed to `0000 1000` inverting the function behaviour). This attack is detected by Property

C, although Property C needs to be located outside the function call so as to check the value of `grantAccess` after the function return (or more precisely the returned value that corresponds to `grantAccess`).

These attacks illustrate that the process and implementation here are not limited to a single fault model or kind of fault injection attack. Thus, the same process and implementation can be used for a variety of fault models and fault injection attacks, as long as some consideration is taken to choose the right properties. Further, the process and implementation here can detect fault injection attacks that cannot be found by checking the source code or intermediate representation alone; the executable binary must be part of the process and implementation.

B. Computational Data

This section discusses the experimental results of automating the the process and implementation of this paper. This automation is straightforward once all the tools are available.

To support the automation experiments, a simple fault injection tool was created[2]. This tool takes an executable binary, and produces mutant binaries by replacing one byte with 0. This fault injection model is naive, but simple to implement and conduct experiments with to test the automation of the process and implementation.

To estimate the feasibility of finding arbitrary fault injection vulnerabilities in a executable, the following experiment was conducted. A script was written[3] that takes a source code and properties written in C, and compiles these to an executable binary with GCC. This executable binary is then validated to ensure the properties hold. The fault injection tool was used to generate mutant binaries for each possible byte in the executable binary being set to 0. The script then enters into a loop over the mutant binaries that generates the LLVM-IR for the mutant binary and model checks the properties on this LLVM-IR. The result of this model checking is then used to determine if the injected fault found a fault injection vulnerability. The runtime and number of fault injection vulnerabilities was counted and reported at the end of the experiment.

This script was run over a version of the motivating example (from Section II). The fault injection tool was limited to injecting faults into the `.text` area of the executable binary that corresponds to the compiled source code (to reduce time wasted model checking fault injection vulnerabilities in the header or other unrelated parts of the executable binary).

The executable binary produced by GCC in this case was 3024 bytes. The `.text` area was 159 bytes and thus 159 1-byte fault injection attacks were simulated, yielding 159 mutant binaries. The following experiments were conducted on a virtual machine configured with one CPU, and 7662 MB of RAM running Linux Ubuntu 14.04 LTS. The virtual machine was hosted on a Macbook Pro with 3,1 GHz Intel Core i7 processor, 16 GB of RAM, running OS X EL Capitan 10.11.

Three different experiments were conducted, testing the three different properties presented earlier. The first experiment with Property A detected 36 fault injection vulnerabilities, and had a runtime of 7 minutes. The second experiment with Property B detected 37 fault injection vulnerabilities, and had runtime of approximately 1 hour. The third experiment with Property C detected 37 fault injection vulnerabilities, and had a runtime of approximately 2 hours. The main cost in time was the model checking by LLBMC, as is clearly shown by the significant difference made by the choice of property.

Observe that the number of fault inject attacks to test in this manner is linear in the size of the binary executable. Thus, automatically testing all such fault injection attacks is feasible, particularly since the implementation can be easily run in parallel.

VII. Case Study: PRESENT

This section presents a case study of five different fault injection attacks against the PRESENT algorithm.

A. Experimental Design

All the experiments tested a single property to capture the capability of a fault injection attack to bypass the encryption algorithm. The property checked whether the "ciphertext" at the end of the encryption was different to the "plaintext". Thus, violations of this property indicated the encryption algorithm had been effectively bypassed. The result of each fault injected mutant binary were thus classified into one of: *passed* where model checking passed; *infinite loop* when the fault caused an infinite loop; *crashed* when the fault caused the program to crash; and *vulnerable* when the fault caused the property to be violated.

The five fault models are: *modifying an unconditional jump* (JMP) to jump to a new address; *modifying a conditional jump* (JBE) to jump to a new address; *zero 1 byte* (Z1B) that sets a single byte to zero; *zero 2 bytes* (Z2B) that sets two consecutive bytes to zero; and

[2]The code for this tool is available upon request.

[3]This script is available upon request.

NOP'ing an instruction (NOP) that sets a byte to a non-operation code. Each is detailed below when considering the results for that fault model.

B. Results Overview

An overview of the results for injecting these fault models in all possible locations in the PRESENT binary can be seen in Table I. All the fault models tested caused crashes, with these being most common with the zero 2 byte and NOP code 1 byte fault models. Infinite loops were also quite common, either through modification of jumps, damaging iterator code, or damaging conditionals. Vulnerabilities were quite rare, which was as expected, with all arising from the jump fault models. The rest of this section considers each of the fault models and the associated experimental results in detail.

	Fault Model					Colour
Result	**JMP**	**JBE**	**Z1B**	**Z2B**	**NOP**	
Passed	1632	502	905	855	784	
Infinite Loop	106	0	53	49	93	
Crashed	62	60	172	225	248	
Vulnerable	1	8	0	0	0	

TABLE I: Overview of Fault Injection Results.

C. Unconditional Jump

The fault model for this experiment was to identify unconditional jump instructions and change their target. For simplicity only increasing the value of the target address was considered (i.e. jumping relatively forward, not relatively backwards). Column **JMP** of Table I presents aggregate results. There are 10 unconditional jumps in the PRESENT binary at addresses 0x0120, 0x014B, 0x0155, 0x018C, 0x01D3, 0x0207, 0x02D4, 0x0313, 0x0361, and 0x0447. The only jump that yielded a vulnerability was at 0x014B, and the details are shown in Figure 6 showing the offsets that could be jumped to and the result of checking each mutant (and the blue box indicating the end of the experiment range).

Most of the significant changes here were infinite loops, with a significant number of crashes, and a single vulnerability that skipped the entire encryption algorithm. The infinite loops are largely as expected, since the modified jump can easily skip loop iterator increment code. The crashes are also to be expected, mostly related to jumping to incorrect byte offsets for the instructions, and so yielding invalid instructions (or instructions that crash in other ways such as trying to read invalid memory segments). The single vulnerability was when the jump for the first loop of the encryption algorithm skips over the entire encryption, going straight to the end of the code. Only a single instance was found as most jumps were "short" (single byte offset), meaning they could not bypass significant amounts of code.

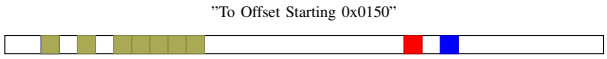

Fig. 6: Unconditional Jumps from Jump at 0x014B

D. Conditional Jump

The conditional jump fault model is very similar to the unconditional jump fault model, changing conditional jump targets in the same manner. Column **JBE** of Table I presents the summaries for the two conditional jumps at addresses 0x02C9 and 0x043C. Again vulnerabilities were only found in one at 0x043C and these are detailed in Figure 7.

Here no infinite loops were detected likely due to the conditions always being triggered at least once, instead only crashes where the unconditional jumps were instead targeting bad locations in the code leading to incorrect "instructions". More interesting are the vulnerabilities that fall into two groups. The first group (the first three in the map) jumped to later assignment instructions (including incorrectly offset locations) that ended up bypassing the correct loop controls (by changing values used for later loop control flow), and eventually skipping the encryption algorithm. The second group (the remaining five) simply jumped to the end of the encryption code, merely bypassing the encryption algorithm.

E. Zero 1 Byte

Another fault model to test automating the process over a larger number of mutants was to set a single byte to zero. There are 1130 bytes in the PRESENT executable binary, and each was set to zero in a different mutant, yielding the results shown in the **Z1B** column of Table I. Detailed results showing which faulted bytes yield which effect can be seen in the map in Figure 8.

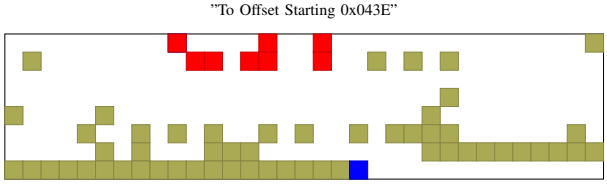

Fig. 7: Conditional Jumps from Jump at 0x043C

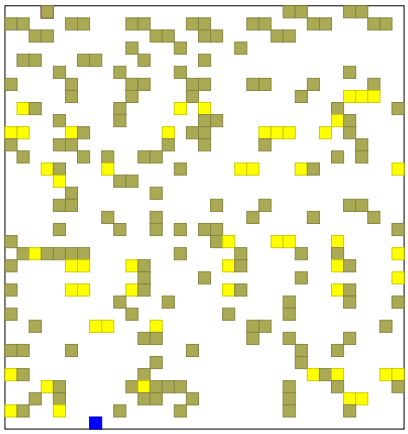

Fig. 8: Zero 1 Byte

Fig. 9: Zero 2 Bytes

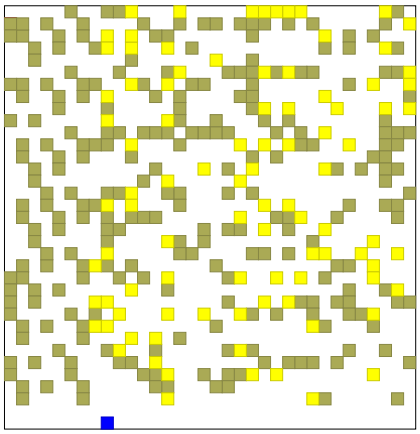

Fig. 10: NOP 1 Byte

No fault injection vulnerabilities to this fault model were detected, although many crashes and infinite loops were introduced. This is not a surprising result, since the PRESENT source code has two top-level loops that both perform some part of the encryption. Thus, although setting one byte to zero could skip either one of these, it would require two (non-consecutive) zero one byte fault injection attacks to be "vulnerable" according to the property tested here.

F. Zero 2 Bytes

A similar test of automation over many mutants was the fault model that sets two consecutive bytes to zero. There are 1129 possible mutant binaries under this fault model. Their results shown in the **Z2B** column of Table I, and the map showing the starting index of the two bytes is shown in Figure 9.

Similar to the zero 1 byte fault injection model, no vulnerabilities were detected. Even more crashes were introduced, although a few less infinite loops. Generally this is due to instructions being damaged to yield failure, either by being simply incomprehensible, or by pushing memory access outside acceptable bounds.

G. NOP Code 1 Byte

The last fault model for this experiment was to set each byte to the instruction code for a non-operation (NOP). This fault injection attack was applied to each of the 1130 bytes to ensure complete coverage (and so in some cases had effects other than NOP'ing an instruction). Column **NOP** of Table I summarises these results, with the detailed map in Figure 10.

This approach turned out to be even more destructive than either of the zero byte fault models. Although more crashes were introduced, the almost doubling of infinite loops was an expected result that could be investigated further in future. No vulnerabilities were detected here which aligns with the prior results that binaries are fairly resistant to these kinds of byte attacks.

H. A Note on Scalability

The experiments were conducted on a variety of devices with different hardware and configurations (all were virtual machines running Ubuntu X64). The distribution was due to different experiments being run at different times, however this makes it impossible to provide consistent runtime information for the experiments.

That said, in general the model checking (either verification or checking) was by far the most expensive in terms of runtime. No attempt was made to optimise or modify the settings of LLBMC to improve runtime, despite some results taking many minutes. This is due to the process being trivially parallisable, since each mutant can be checked independently.

VIII. Related Work

This section discusses related work and their differences with respect to the process presented here.

One of the first uses of formal methods to analyse fault injection vulnerabilities was to verify a counter measure in the implementation of CRT-RSA by analysing the C source code [9]. The authors show that by adding ANSI/ISO C Specification Language (ACSL) [21] properties to the CRT-RSA pseudocode, they could verify that the Vigilant's CRT-RSA countermeasure sufficiently protects against fault injection attacks. The lack of fault injection and analysis on the binary limits the attacks that can be detected to those that have a representation in the source code. Further, the analysis was only for a single countermeasure to prove it worked, rather than to

consider all possible fault injection attacks and models as is the goal of the process presented here.

Perhaps the closest to the process presented here is the Symbolic Program Level Fault Injection and Error Detection Framework (SymPLFIED) [22] a program-level framework to identify potential vulnerabilities in a software. The vulnerabilities are detected by combining symbolic execution and model checking techniques. However, the SymPLFIED framework is limited as SymPLFIED only supports the MIPS architecture [23]. One of the proposed future work in [24] was building front-end to support X86 architecture, but to the best of our knowledge, no further work has been published on supporting new architectures in SymPLFIED.

In [25], Potet et al. present Lazart, a tool that can simulate a variety of fault injection attacks and detect vulnerabilities using formal methods. The Lazart process begins with the source code which is compiled to LLVM-IR. The simulated fault is created by modifying the control flow of the LLVM-IR. Symbolic execution is then used to detect differences in the control flow, and thus detect vulnerabilities. Although this high level approach is similar to that of this paper, Lazart is unable to reason about or detect fault injection attacks that operate on binaries rather than the LLVM-IR. Further, the choice of symbolic execution does not account for concrete values, and so is less complete than model checking [26], [27].

In [11] the authors propose combining the Lazart process with the Embedded Fault Simulator (EFS) [28]. This extends from the capabilities of Lazart alone by adding lower level fault injection analysis that is also embedded in the chip with the program. The simulation of the fault is performed in the hardware, so the semantics of the executed program correspond to the real execution of the program. However, EFS is limited to only considering instruction skip faults (equivalent to NOPs of Section VII-G).

An entirely low level approach is taken by Moro et al. [10] who use model checking to formally prove the correctness of their proposed software countermeasures schemes against fault injection attacks. The approach has some similarities to here: using model checking while focusing on low level representations. However, the focus is on a very specific and limited fault injection model that causes instruction skips and ignores other kinds of attacks. Further, the model checking is over only limited fragments of the assembly code, and not the program as a whole.

A less formal approach is taken in [29] where experiments are used for testing the TTP/C protocol in the presence of faults. Rather than attempting to find fault injection attacks, they injected faults to test robustness of the protocol. They combined both hardware testing and software simulation testing, comparing the results as validation of their approach.

A fault model inference focused approach is taken by Dureuil et al. [30]. They fix a hardware model and then test various fault injection attacks based upon this hardware model. Fault detection is limited to EEPROM faults on the ARMv7-M architecture. The fault model is then inferred from the parameters of the attack and the embedded program. The faults are simulated upon the assembly code and the results checked with predefined oracles on the embedded program. Although the approach uses neither formal methods nor general fault models, the choice of fault model and derivation of this provides some interesting guidance for selecting fault models and future work.

IX. Conclusions and Future Work

Fault injection has recently been increasingly used to attack software applications, and test system robustness. This paper presents a formal process that uses model checking to detect fault injection vulnerabilities in binaries. This process supports the detection of many varieties of fault injection vulnerabilities, and does not rely on any particular system architecture, fault model, or other restricted choices (as are common in the literature).

Experimental results demonstrate the efficacy or the process by testing a variety of fault models on the PRESENT binary. For naive fault models the (zero 1 byte, zero 2 consecutive bytes, NOP code 1 byte) this yielded many infinite loops and program crashes, but no attacks to skip the entire algorithm. However, when (both unconditional and conditional) jump instructions were targeted, 9 fault injection vulnerabilities were found that allow complete bypassing of the PRESENT algorithm.

A. Future Work

There are several limitations with the implementation chosen here that provide opportunities for future research. Indeed the implementation used was merely the easiest to combine effectively to implement the process.

The choice to use MC-Sema was to be able to work with LLVM-IR. The choice of LLVM-IR is due to this being a widely used intermediate representation that is supported by many other tools. However, there are limitations with MC-Sema that may limit future work with the implementation used in this paper. MC-Sema supports only (some of) the X86 architecture [18] and so future work is to replace MC-Sema and/or expand to implement the process for other architectures.

The LLBMC model checker is sufficient for the safety properties but does not support liveness properties. Thus although LLBMC was sufficient for the proof of concept here, future work will exploit a non-bounded model checker that can also accept liveness properties. In particular a model checker that can produce traces (LLBMC can, but not combined with MC-Sema) would aide in understanding vulnerabilities and analysing results.

Fault injection was implemented with custom tools for this work, although there already exist several tools to simulate fault injection attacks on software [31], [32]. However, these tools are limited by various choices that make unsuitable for the process here (hence their lack of use in the implementation). Several are only able to inject faults into intermediate representations, and not into executable binaries [33], [25], [34], thus being unable to simulate faults that appear only at the executable binary level. Others have different limitations, such as: specific hardware platforms [22], [35], specific source code languages [36], [9], [34], or requiring simulating drivers [37]. Despite these limitations, many include useful techniques or developments that could be incorporated into future development of a general fault injection tool for executable binaries.

Complementary research is to explore ways to inject faults intelligently. This could exploit knowledge of the property to inject faults that would lead to property violations, yielding improved efficiency of experiments.

Regarding properties, another area of future work is to consider how to extract properties automatically from the binary (or source code). There is some existing work in this area [38], [39] although they focus upon high level behaviour rather than binary code.

Currently the process identifies vulnerabilities, but does not suggest fixes or countermeasures. Automatically generating countermeasures is non-trivial, although if countermeasures to particular faults are known future work could suggest or implement them automatically. Perhaps more significantly, these countermeasures could be checked immediately using the process here and so their effectiveness verified immediately.

There are also several directions related to case studies. The analysis of PRESENT here was proof-of-concept to demonstrate the process, and indeed some of the "passed" results here correspond to known differential attacks against PRESENT [40], [41]. Applying the process with more properties to consider, and a better suite of fault models would yield more complete results on the vulnerability of the PRESENT binary.

Future case studies could consider other security critical software, e.g. encryption algorithms, mission critical software, embedded device kernels, and also software that has implemented countermeasures.

REFERENCES

[1] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The sorcerer's apprentice guide to fault attacks." *IACR Cryptology ePrint Archive*, vol. 2004, p. 100, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/iacr/iacr2004.html\#Bar-ElCNTW04

[2] S. Guilley, L. Sauvage, J.-L. Danger, and N. Selmane, "Fault injection resilience," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2010 Workshop on*. IEEE, 2010, pp. 51–65.

[3] I. Verbauwhede, D. Karaklajic, and J.-M. Schmidt, "The fault attack jungle-a classification model to guide you," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2011 Workshop on*. IEEE, 2011, pp. 3–8.

[4] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3. IEEE Press, 2014, pp. 361–372.

[5] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black Hat*, 2015.

[6] K. S. Yim, "The rowhammer attack injection methodology," in *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*. IEEE, 2016, pp. 1–10.

[7] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache, "Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3056–3076, 2012.

[8] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz, "Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*. IEEE, 2013, pp. 77–88.

[9] M. Christofi, B. Chetali, and L. Goubin, "Formal verification of an implementation of CRT-RSA vigilant's algorithm," in *PROOFS Workshop: Pre-proceedings*, 2013, p. 28.

[10] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson, "Formal verification of a software countermeasure against instruction skip attacks," *Journal of Cryptographic Engineering*, vol. 4, no. 3, pp. 145–156, 2014.

[11] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys, "Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks," in *International Symposium on Foundations and Practice of Security*. Springer, 2014, pp. 92–111.

[12] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe, "Present: An ultra-lightweight block cipher," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2007, pp. 450–466.

[13] L. R. Knudsen and G. Leander, "Present–block cipher," in *Encyclopedia of Cryptography and Security*. Springer, 2011, pp. 953–955.

[14] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[15] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith, "Proactive detection of computer worms using model checking," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 424–438, 2010.

[16] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," *Advances in computers*, vol. 58, pp. 117–148, 2003.

[17] B. Gough, *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[18] Trail of bits, "Mc-semantics," 2016, https://github.com/trailofbits/mcsema.

[19] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded model checking of C and C++ programs using a compiler IR," in *Proceedings of the 4th International Conference on Verified Software: Theories, Tools, Experiments*, ser. VSTTE'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 146–161. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27705-4_12

[20] C. Sinz, F. Merz, and S. Falke, "LLBMC: A bounded model checker for LLVM's intermediate representation - (competition contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, 2012, pp. 542–544. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28756-5_44

[21] P. Baudin, J.-C. Filliatre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C specification language, version 1.4," *CEA*, vol. 6, 2009. [Online]. Available: http://frama-c.com/download/acsl_1

[22] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer, "SymPLFIED: Symbolic program-level fault injection and error detection framework," in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 2008, pp. 472–481.

[23] C. Price, "MIPS iv instruction set," 1995.

[24] F. Q.-Y. Yuan, "Formal framework and tools to derive efficient application-level detectors against memory corruption attacks," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2010.

[25] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil, "Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 213–222.

[26] C. S. Păsăreanu and W. Visser, *Symbolic Execution and Model Checking for Testing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 17–18. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-77966-7_5

[27] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 654–665.

[28] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant, "Idea: embedded fault injection simulator on smartcard," in *International Symposium on Engineering Secure Software and Systems*. Springer, 2014, pp. 222–229.

[29] A. Ademaj, P. Grillinger, P. Herout, and J. Hlavicka, "Fault tolerance evaluation using two software based fault injection methods," in *On-Line Testing Workshop, 2002. Proceedings of the Eighth IEEE International*. IEEE, 2002, pp. 21–25.

[30] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière, "From code review to fault injection attacks: Filling the gap using fault model inference," in *International Conference on Smart Card Research and Advanced Applications*. Springer, 2015, pp. 107–124.

[31] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, Apr. 1997. [Online]. Available: http://dx.doi.org/10.1109/2.585157

[32] A. Johansson, "Software implemented fault injection used for software evaluation," *Building Reliable Component-Based Systems*, 2002.

[33] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," in *Workshop on Silicon Errors in Logic System Effects (SELSE)*, 2013.

[34] S. K. Vishal Chandra Sharma, Ganesh Gopalakrishnan, "Towards reseiliency evaluation of vector programs," in *21st IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems (DPDNS)*, 2016.

[35] L. Riviere, Z. Najm, P. Rauzy, J.-L. Danger, J. Bringer, and L. Sauvage, "High precision fault injections on the instruction cache of ARMv7-M architectures," in *Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 62–67.

[36] J.-B. Machemie, C. Mazin, J.-L. Lanet, and J. Cartigny, "SmartCM a smart card fault injection simulator," in *2011 IEEE International Workshop on Information Forensics and Security*. IEEE, 2011, pp. 1–6.

[37] K. Cong, L. Lei, Z. Yang, and F. Xie, "Automatic fault injection for driver robustness testing," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 361–372.

[38] F. Rogin, T. Klotz, G. Fey, R. Drechsler, and S. Rulke, "Advanced verification by automatic property generation," *IET computers & digital techniques*, vol. 3, no. 4, pp. 338–353, 2009.

[39] Y. Zhu and H. Gao, "A novel approach to generate the property for web service verification from threat-driven model," *Appl. Math*, vol. 8, no. 2, pp. 657–664, 2014.

[40] G. Wang and S. Wang, "Differential fault analysis on present key schedule," in *Computational Intelligence and Security (CIS), 2010 International Conference on*. IEEE, 2010, pp. 362–366.

[41] N. Bagheri, R. Ebrahimpour, and N. Ghaedi, "New differential fault analysis on present," *EURASIP Journal on Advances in Signal Processing*, vol. 2013, no. 1, p. 145, 2013.